DEFENCE
SCIENCE & TECHNOLOGY

Situation Description Language Implementation

S. Greenhill and S. Venkatesh and
A. Pearce and T.C.Ly

DSTO-GD-0342

Situation Description Language Implementation

S. Greenhill and S.Venkatesh

Curtin University of Technology

A. Pearce

University of Melbourne

T.C. Ly

Maritime Operation Division

Systems Sciences Laboratory

DSTO–GD–0342

ABSTRACT

SDL is a Situation Description Language intended for use in situation assessment problems. SDL provides knowledge modelling and inference facilities for reasoning with information.

This document describes a portable implementation of SDL in Java. It provides information required by a user of the system. Details include the operation of the compiler, the use of temporal knowledge and inference, and use of the visualisation system. This report also provides implementation details necessary for modifying or extending the system. A detailed example describes how the system was used for submarine situation assessment.

APPROVED FOR PUBLIC RELEASE

Published by

DSTO Systems Sciences Laboratory
PO Box 1500
Edinburgh, South Australia, Australia 5111

Telephone: *(08) 8259 5555*
Facsimile: *(08) 8259 6567*

AR No. 012-486
November, 2002

APPROVED FOR PUBLIC RELEASE

Situation Description Language Implementation

EXECUTIVE SUMMARY

Situation assessment is an essential process prior to making a decision. On submarines and other military platforms the operators take information from available sensors and their background knowledge to deduce the tactical situation. Designing systems to replicate this process will give a better understanding of the process itself, and opens the possibility of automating the tasks that computers perform better. The process is information intensive, requiring a high level language with concepts like those found in the field of artificial intelligence. The development of the Situation Description Language (SDL) brings together various techniques useful for representing the assessment process. The realisation of SDL provides an actual working language that not only encodes the process, but enacts it.

This report describes how the SDL was implemented as a compiler and interpreter under Java, which will be referred to as a Situation Assessment Processor (SAP). As a Java program it is portable on any platform on which a Java virtual machine exists. The core of implementation makes use of the RETE algorithm to ensure efficient processing of rules. The temporal reasoning allows data to be associated with a precise time, or at some abstract point on the timeline.

Concepts that are important to a domain only become clear after obtaining a detailed understanding of the domain from experts. The SAP leverages Java to allow seamless integration of new objects on top of existing SDL featues. These objects will embody those new concepts. For example, spatial objects and their operations are concepts that were added to SDL this way.

The resultant situation assessment can be sent to third party software. A subscription mechanism exists to ensure only the required information reaches the third party software, and not swamp it with unnecessary information. Alternatively, results can be displayed using the integrated visualisation facilities. These facilities allow displaying of spatial information, and temporal information as it is created. Having experts view the encoded expertise in action enables them to give feedback as to the validity of the encoding.

The SAP was demonstrated by encoding a limited Submarine Situation Assessment. It shows how SDL was able to encode the required temporal and non temporal knowledge, and the rules that manipulate this knowledge. It also shows how multiple hypotheses enable different interpretation of a situation by varying the assumptions.

Contents

Figures

Tables

1 Overview

This document describes the implementation of SDL, a Situation Description Language[1]. The needs for SDL came after looking at what was available at the time [4]. SDL is a language intended to model knowledge within simulation environments, and provide a framework for reasoning with this information. A companion document gives a formal definition of the language [5].

It includes:

- Object-oriented data modelling with support for type-bound procedures and single-inheritance.
- A forward-chaining inference system with RETE-based pattern matcher.
- A procedural programming system loosely based on the language Oberon-2.
- Representations for time and space.
- Representations for uncertainty, including a system for handling multiple concurrent hypotheses.
- Parameterised types for sets and sequences.

SDL is currently implemented as a compiler and interpreter written in the Java language. Source code written in SDL is compiled to a form of abstract syntax tree which is interpreted to run the program. SDL programs can call methods of Java objects; the compiler dynamically loads Java classes as required by SDL programs.

SDL is a strongly typed language so many semantic errors can be identified in the compilation phase. A program is only executed after being successfully compiled. If run-time errors occur during execution, the program halts with a description of the error and shows the location in the source code and the state of program variables.

2 Using the Compiler

The SDL system includes both a compiler and interpreter. The compiler is invoked using the command:

```
java SDL.Comp {options} {sourceFiles}
```

One or more source files may be given. The system searches for imported modules within the directory of the importing module. There are two situations where input may be read from objects other than files: standard input and named pipes. In these cases, imported modules are taken from the current directory. The file name `stdin` causes input

[1]SDL not related to the "Specification and Description Language" SDL standardized as ITU (International Telecommunication Union) Recommendation Z.100

to be read from standard input. A file name beginning with the pipe prefix (\\.\pipe\ under Windows) is treated as a named pipe.

Currently, the following options are supported, mainly for debugging purposes:

-n Enable RETE Network view. This option causes SDL to open a window showing a graphical view of the RETE network resulting from compiling rules. The user may click on the nodes to discover the attributes of nodes, and any tokens in the node output set.

-tactiv Traces the activation of rules. Rule firing is initiated by `System.Run`, but rules may be activated / deactivated many times outside of an inference cycle. This feature allows the user to determine when a rule is activated / deactivated in response to USER statements.

-tfire Traces the firing of rules. During `System.Run`, the system fires rules from a set of active rules until no further active rules remain. With this option enabled, the system outputs the rule name and token associated with each rule firing.

-tsimple Use simple time model (see section 6).

-ttime Reports the run-time for each USER statement.

-tresolve Traces binding resolution in the RETE compiler.

-tapply Traces application of tests at each node in the RETE network.

-tprop Traces propagation of tokens into each RETE network node.

-tgraph Trace temporal graph operations.

-thypothesis Trace hypothesis cloning.

-timport Show Java class imports.

-s Prints the compiler scope before terminating.

-h Shows a list of options.

SDL includes USER declarations, which identify statements that are issued interactively by the user. Each statement in a USER declaration is executed immediately before the next statement is compiled.

Example:

```
  java SDL.Comp stdin
  USER
  PRINTLN 1 + 2 * 3 + 4;
> 11
  PRINTLN "Hello " + "There";
> Hello There
  PRINTLN SIZE("Hello " + "There");
> 11
  END.
```

The USER statement allows interactive statements to be used with pre-compiled code. For example:

```
  java SDL.Comp test/sieve.sdl stdin
  USER
  PRINTLN Primes();
> stdin:2,15: Cannot find matching procedure declaration : Primes
> stdin:2,16: Designator has no return type :
  PRINTLN Primes(20);
> {13, 11, 7, 5, 3, 19, 2, 17}
  PRINTLN IntRange(1, 20);
> {20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1}
  PRINTLN IntRange(1,20) - Primes(20);
> {20, 18, 16, 15, 14, 12, 10, 9, 8, 6, 4, 1}
  END.
```

The SDL system includes a simple debugger that shows the state of the program when a run-time exception occurs. For example, the following program contains an error.

```
MODULE Error1;

TYPE
  T = RECORD x : INTEGER END;

VAR
  t : T;

PROCEDURE (t : T) Put(x : INTEGER);
BEGIN
  t.x := 100 DIV x; (* ERROR when x = 0 *)
  t.Put(x-1);
END Put;

BEGIN
  t := NEW T(:x 1);
  t.Put(3);
END Error1.
```

The procedure Put is called recursively and will eventually cause a divide-by-zero exception. Running the program causes the following:

```
  java SDL.Comp test/Error1.sdl stdin

--------------SDL State-------------------------
PROCEDURE Put (Error1.sdl:14,11)
 t = T(:x* 100, :tag T0)
```

```
 x = 0
PROCEDURE Put (Error1.sdl:14,11)
 t = T(:x* 100, :tag T0)
 x = 1
PROCEDURE Put (Error1.sdl:14,11)
 t = T(:x* 100, :tag T0)
 x = 2
PROCEDURE Put (Error1.sdl:19,9)
 t = T(:x* 100, :tag T0)
 x = 3
GLOBAL VARIABLES
 System = SDL.SystemObject@297b0b
MODULE Error1
 t = T(:x* 100, :tag T0)

Run Time Error: Division by zero (Error1.sdl:13,18)

10:
11: PROCEDURE (t : T) Put(x : INTEGER);
12: BEGIN
13:   t.x := 100 DIV x; (* ERROR when x = 0 *)
----------------------^ Division by zero
14:   t.Put(x-1);
15: END Put;
16:
Processing stdin
```

SDL prints a back-trace showing the values of global variables and local variables in each procedure activation. It also gives the approximate source location where the error occurred. In the example, `Error1.sdl:13,18` means column 18 of line 13 of the file `Error1.sdl`.

This debugger is intended to allow errors to be easily located. When a run-time error occurs execution stops. Any active procedure activation records are removed from the stack, but global variables and objects are unaffected. In an interactive session, the user may continue using the system following a run-time error, although the system state may not be well-defined.

3 Java Interface

Java provides a standard "reflection" service which enables a program to inspect its own type system at run-time. The SDL compiler uses the `java.lang.reflect` services to dynamically load Java classes and to generate SDL bindings for Java types and methods.

Java classes can be used within SDL with the following restrictions:

- Only Java class methods are available. It is not possible to use static procedures, or variables.
- Exceptions are not supported. If a Java class method throws an exception, the program will be terminated.
- A restricted set of types are available. Java types that correspond directly to SDL types may be used. Currently, SDL does not support any methods that would require translation of values at run-time.[2]

  The table below shows the correspondence between SDL types and the allowed types in Java methods. Note that arrays are not supported.

Java Type	SDL Type
`int`, or `java.lang.Integer`	`INTEGER`
`java.lang.String`	`STRING`
`boolean`, or `java.lang.Boolean`	`BOOLEAN`
`double`, or `java.lang.Double`	`REAL`
Object type `T`	`JAVA T`

Internally, SDL uses Java interfaces to provide some of its services. These include:

- The `System` object (see 5).
- Some primitive data types. These include `SPATIAL` (`SDL.SpatialObject`), `POINT` (`SDL.SpatialPoint`), `LINE` (`SDL.SpatialLine`), and `Hypothesis` (`SDL.Hypothesis`).

There are a few differences between the type systems of SDL and Java that can affect the usefulness of Java interfaces. Some SDL constructs have no corresponding equivalent in Java, so it will not always be possible to obtain the desired method signature in a pure Java declaration. For example, SDL has a parametric SET type whereas Java sets have a single member class `java.lang.Object`.

One way around this problem is to use SDL `HINT` declarations. These inform the SDL compiler of the correct interpretation for methods. For example:

```
HINT SDL.SpatialLine (Intersect) : SET OF SPATIAL;
```

This informs the SDL compiler that the return type of the `Intersect` method of the `SDL.SpatialLine` class is `SET OF SPATIAL`. Without this declaration, the result type determined from the method signature would be `java.util.Set`.

Currently the `HINT` mechanism is restricted to return type declarations, although it could usefully be extended to handle the whole method signature including method parameters.

The following example illustrates the use of Java classes within SDL code.

[2]Translating values at run-time could be implemented, but would incur a performance penalty. Internally SDL uses java.lang types for its own primitive types so it is currently not necessary to translate any values in order to call a Java method.

```
PROCEDURE FetchURLString (name : STRING) : STRING;
(* Attempt to fetch data from URL <name>. The resulting text is returned
 * as a STRING *)
TYPE
  StringBuffer = JAVA java.lang.StringBuffer;
  InputStream = JAVA java.io.InputStream;
  URL = JAVA java.net.URL;
VAR
  i : InputStream;     (* input from server *)
  c : INTEGER;         (* character read from input *)
  s : StringBuffer;    (* result string *)
BEGIN
  (* open stream of input from URL *)
  i := NEW URL(name) .openConnection() .getInputStream();
  s := NEW StringBuffer();

  (* fetch data from the stream, and append to string *)
  c := i.read();
  WHILE c # -1 DO
    s := s.append(CHR(c));
    c := i.read()
  END;

  i.close();
  RETURN s.toString();
END FetchURLString;
```

4 Interpreter

SDL compiles source code to a form of abstract syntax tree. Each node in the tree contains:

- Enough semantic information to interpret each statement at run-time.
- References to the source code context so that accurate debugging information can be given if errors occur at run-time.

`SDL.Node` is the abstract root class for all tree nodes. This class contains a source token reference and some exception handling.

`SDL.Expr` is an abstract subclass of `SDL.Node` which encodes all language constructs that return values (eg. expressions, operators, function calls). Every `SDL.Expr` has a result type, and an evaluation function:

```
abstract Object Eval(Env e) throws RunTimeError;
```

To evaluate an expression, the interpreter calls `Eval` passing an environment of type `SDL.Env`. The environment represents all context information. This includes:

- Global variables.
- Local variables and parameters represented using a stack of procedure activation records.
- Results of procedure RETURNs. The expression that calls a function is generally not the parent of the expression that returns the function result (eg. a RETURN may be nested in several levels of iteration). These intermediate activations must be unwound so that the return value can be propagated. State variables in the environment control this process.
- Pattern bindings. Expressions that occur in rules contain references to symbols that designate pattern variables. These expressions are only evaluated during token propagation.
- The temporal graph.
- Any information that controls the results of expression evaluation. In particular, state information is maintained to support the UPDATE statement.

`SDL.Stat` is an abstract subclass of `SDL.Node` which encodes all language constructs that do not return values (ie. statements).

```
abstract void Exec(Env e) throws RunTimeError;
```

`SDL.Stat.Exec` is similar to `SDL.Expr.Eval` except that it does not return a result.

The interpreter can be extended to handle new language constructs. The general procedure is:

1. Implement one or more subclasses of `SDL.Expr` or `SDL.Stat` to handle the semantics of the construct at run-time.
2. Implement one or more procedures within `SDL.Compiler` to create tree nodes after checking for correct semantics.
3. Adjust the attributed grammar `SDL.atg` to parse language statements. Generally, code in `SDL.atg` should be as simple as possible; any complex code should be in `SDL.Compiler`.

5 The System object

Some system-level functions are exposed via the `System` object. This is a pre-defined object implemented in `SDL.SystemObject`. Important functions are:

`System.Run()` This activates the inference system, which will continue activating rules until no further rules are satisfied.

`System.Stats(level)` Prints run-time statistics, including the number of rules activated / deactivated, and the number of objects created / deleted. If `level=0`, just summary information is printed. If `level=1`, detailed (per class / rule) information is printed.

`System.ShowEnv()` Prints the system execution context. This includes the values of global and local variables. This may be called during a procedure to help debug the context of the procedure call without halting the program.

`System.GC()` This calls the system garbage collector. This is not normally required, but may be useful during debugging.

`System.MainView()` This creates or returns the main system viewer, an object of type `SDL.MainViewInterface` (see 8.1.3).

`System.TestInit()` This initialises the notification system (see 9) for test output. Notification messages are sent to System.out.

`System.DMarsInit()` This initialises the notification system to use an agent within the DMars system. Note the dMars host parameters are defined in `SystemObject.java`.

`System.Notify()` This notifies a client of the current state of the system. Any registered events will be sent to the client, and the client may issue queries to SDL.

`System.Active("rule")` displays active bindings for the named rule.

`System.Active("")` displays active bindings for all rules. Both this and the above form display bindings for all activation tokens: those that have and have not been used to fire the rule.

`System.Active("rule",false)` displays only those bindings that have not already fired the rule.

6 Temporal Knowledge

The SDL system handles time in two ways. Precise times may be represented using REAL/INTEGER values. Imprecise or qualitative temporal knowledge may be represented using abstract *points* in time. A *temporal constraint* is a relationship between points in time.

The SDL temporal representation corresponds to a simple temporal problem (STP) [2]. In this representation, a set of variables $X_1, ..., X_n$ represent points in time. A constraint is an edge E_{ij} labelled by an interval $[a_{ij}, b_{ij}]$, which represents the constraint:

$$a_{ij} \leq X_j - X_i \leq b_{ij}$$

Alternatively, the constraint may be viewed as a pair of inequalities:

$$\begin{aligned} X_j - X_i &\leq b_{ij} \\ X_i - X_j &\leq a_{ij} \end{aligned}$$

The solution to a STP can be constructed by applying *Floyd-Warhsall's* all-pairs-shortest-paths algorithm. This algorithm runs in time $O(n^3)$ and detects inconsistencies in the constraint network. This computes the minimal a_{ij} and b_{ij} for all $1 \leq i, j \leq n$.

Generalised temporal constraint satisfaction problems (TCSP) allow edges to be labelled by several intervals. These are currently not handled by SDL. The solutions to TCSP are well described [2], but are known to be NP-hard.

The need to handle hypotheses introduces some complexity into the SDL temporal model. Each hypothesis corresponds to a possible world in which a set of hypothetical objects exist. The temporal model includes one STP that expresses constraints between all non-hypothetical points in time. This is called the *root partition* and includes the variables $X(0)_i$ and edges $E(0)_{ij}$. In addition, each hypothesis introduces its own variables $X(k)_i$ and edges $E(k)_{ij}$. For each hypothesis k, there is a corresponding STP defined by variables $X(0) \cup X(k)$ and edges $E(0) \cup E(k)$. If there are m hypotheses, there are $m + 1$ STPs.

In the current implementation, SDL checks for consistency of its temporal knowledge base whenever temporal assertions are added. Addition of assertions within hypotheses therefore require the solution of one STP. Addition of non-hypothetical assertions require the solution of m STPs.

Possible future optimisations are:

- Defer the checking of consistency on addition of edges. This reduces the ability to localise inconsistencies (ie. to determine which statement caused an inconsistency) in favour of better performance.
- Compute solutions incrementally. If a solution already exists it is possible to compute an amended solution in less time than it takes to recompute a full solution.

In situations where only precise temporal knowledge is used, SDL provides a "simple" temporal model. This mode is used when the compiler is invoked with the "-simple" option. Restrictions in the simple model are:

- Only the following constraints are allowed:

  ```
  HAPPENS AT
  STARTS AT
  ENDS AT
  ```

  Temporal node properties are unified when `AT` constraints are asserted. At least one of the constrained nodes must have a defined absolute time.

- Other constraints are not allowed; asserting these constraints generates a run-time exception. No system constraints are asserted (eg. end of interval occurring after start of interval).

- Temporal queries on unconstrained nodes return FALSE (as in normal temporal model). REL applied to unconstrained node throws run-time exception.

- Constraints are not propagated (other than `AT`) and the distance matrix is never computed. Since this process is $O(n^2)$ space and $O(n^3)$ time, the result is significantly faster.

- In addition to simpler TemporalGraph operations, the simple model also affects how temporal changes are propagated. In the normal temporal model, temporal assertions about an event may affect the relationship between any other temporal points, so knowledge about all events must be invalidated when an assertion is made. In the simple model, this does not occur. Only the object that is the subject of the assertion is invalidated. This greatly reduces the "inference load" resulting from temporal assertions. Note: a better approach would be to determine which points have changed relationships, but this optimisation has not yet been implemented.

7 Inference System

The SDL system includes a forward-chaining inference system based on the RETE algorithm [3]. RETE is a generalised algorithm for systems that must match many objects against many patterns.

- *RETE avoids iterating over objects.* A naive approach to pattern matching compares each object with a pattern to determine which objects match. RETE avoids this iteration by storing with each pattern a list of objects which match the pattern. The lists are updated when objects change.

- *RETE avoids iterating over patterns.* A pattern is a set of conditions which are logically combined. RETE avoids iterating over patterns by decomposing a set of patterns into a network in which each element is represented only once, and is shared between all instances of patterns containing that element.

The SDL RETE implementation is based on that described by Forgy for the OPS5 production system [3]. Similar methods are used in the CLIPS expert system shell [1], and in many commercial systems. One limitation of the OPS5 and CLIPS approaches is that they only allow left-associative joins in patterns. Later generalisations of OPS5 have relaxed this restriction, but introduces some complexities into the management of the RETE network at run-time.

SDL handles generalised RETE nets (permitting both left- and right-associative joins) and uses techniques outlined by Lee and Schor [7].

Readers are referred to the literature for further details of RETE implementation.

There are six different types of nodes in an SDL RETE network. Tokens are pushed into the top nodes of the RETE network. When a token influences the result set of a node, it is propagated to its successor nodes. "Join" nodes involve relations between two or more objects. The outputs of join nodes are composite tokens that are the concatenation of their inputs. In this way, tokens flow down the network until they ultimately reach an activation node. Tokens grow longer as more objects are involved in the specification of a pattern. Each rule in the SDL system has a single activation node which is associated with a series of statements to be executed by that rule.

CLASS nodes. These nodes are the root nodes for patterns. Every SDL record type that occurs in a rule has an associated CLASS node. When changes occur to record values, a "Remove" token is first propagated to remove any activations associated with the old value of a record. Then, an "Add" token is propagated to reflect the new value of the record. The output token set of a CLASS node is the set of instances of the class.

ALPHA nodes. These nodes are used to evaluate conditions that apply *within* a pattern. Such conditions can be evaluated without reference to other patterns. ALPHA nodes always have CLASS nodes as inputs. The RETE compiler merges all conditions occurring in multiple rules so that they are only evaluated once for a given record value. The output token set of an ALPHA node is the set of input tokens (class instances) that match the associated conditions.

BETA (join) nodes. These nodes evaluate conditions that apply *between* patterns. A BETA node involves a relationship between two nodes. Its output tokens are concatenated pairs of input tokens that match the associated conditions. In general, each pattern in a rule involves a corresponding BETA node.

ELEMENT nodes. These nodes expand attributes of token elements that have multiple values (eg. SET, SEQUENCE, or POTENTIAL values). An ELEMENT node propagates a token for each member of a multi-valued attribute that matches the associated conditions.

NOT nodes. These nodes are specialised join nodes to handle negated patterns. For each left token it maintains a count of how many right tokens allow a match based on the associated conditions. When the count is zero, the left token is added to the output set. When the count becomes non-zero, the left token is removed from the output set. This is normally trigger by the addition of tokens to the right input.

ACTIVATE nodes. These nodes are terminal nodes in the RETE network. Each ACTIVATE node is associated with the statement body of a rule. The output of the predecessor of an ACTIVATE node is the set of tokens that may activate the rule.

The inference system is invoked by `System.Run()`. The current rule scheduler (`SDL.Compiler.FireRule`) simply searches for the first rule node that has an unused token in its activation set. It marks the token as used, and then executes the associated rule body using the token for pattern bindings. This process continues until no rule has an unused token in its activation set.

The implication of this scheduling strategy is that a rule only fires once for each set of records that match the pattern. If a record involved in a pattern changes, the rule may fire again *even if its satisfiability has not changed.* This is because any change to a record involves a retraction of all information about the record. Conversely, if a rule involves negation it may only fire again *if the satisfiability of the negation has changed.* This is a result of RETE semantics, and similar behaviour is seen in other RETE-based expert systems like CLIPS[1], and JESS[6].

7.1 Rule rewriting

SDL employs a two-stage rewriting process when compiling rules. In the first phase, OR operations are replaced by negated ANDs. Thus:

```
A | B
```

becomes

```
~ (~A & ~B)
```

In the second phase, negated conditions are rewritten as NOT-join nodes. If the negation appears on the right hand side of an AND condition, the BETA node is trivially converted to a NOT node. If the negation appears on the left hand side of an AND, or the negation is a singleton condition in a rule, the system generates a NOT-join node with the negation on the right and with a dummy pattern (of type InitialCondition) on the left hand side.

7.1.1 Event rules

Event Rules are constructed from EVENT pattern condition, a WHEN condition, an optional ACTIVE clause and an optional INACTIVE clause. Event rules are implemented as a pair of Forward rules.

```
RULE Name
EVENT
  Pattern
WHEN
  Condition
END Name;
```

becomes:

```
RULE AssertName
IF
  Condition & ~ Pattern
```

```
THEN
  CREATE Pattern
END AssertName;

RULE RetractName
IF
  Pattern p & ~ Condition
THEN
  DELETE p;
END RetractName;
```

When the Pattern is created, bindings from the WHEN condition are used. If no bindings exist (eg. for a negated condition) an error is signalled.

The ACTIVE/INACTIVE clauses are used to establish temporal constraints on the event as the event is activated / de-activated. ACTIVE constraints are asserted after the object is created in the Assert rule. INACTIVE constraints are asserted before the object deactivated in the Retract rule.

There are two types of events: persistent and non-persistent. Deactivation of events is handled differently for these types. Non-persistent events are DELETEd when the retraction rule fires. Persistent events are not DELETEd, but are made "inactive" meaning that they can no longer fire their EVENT Assert/Retract patterns, but may match other rules. In particular, they may be used in temporal constraints to allow the system to reason about the past.

Persistent events are implemented by adding an "active" field to the EVENT object. This field is initialised to "TRUE" when the object is created. All EVENT patterns are implicitly qualified with "active TRUE" meaning that they only fire when the event is active. The user is free to qualify patterns with "active TRUE" to match active events, "active FALSE" to match inactive events, or to leave the value of "active" unbound, which will match both active and inactive events. The retraction pattern for a persistent event sets "active := FALSE" after asserting any temporal constraints specified in the "INACTIVE" clause.

7.2 Temporal reasoning in rules

Temporal assertions may have wide-ranging consequences. Depending on constraints, changes in the relationship between events may propagate to affect relationships between other events.

This is significant in rules because patterns that match events and may include temporal expressions. Whenever the truth of a temporal expression may have changed, such rules need to be reevaluated for satisfiability.

Currently, SDL employs a "naive" strategy for managing temporal change. Whenever a temporal assertion occurs SDL reevaluates any rules containing events. It does this by propagating a "remove" token followed by an "add" token for each event.

A better approach would be to compute a new solution for each STP, and then compare the old and new solutions to determine which events actually have changed relationships with other events. These events would then be "removed" under the old solution and "added" under the new solution.

8 Standard Packages

8.1 The Visualisation System

SDL includes a set of Java modules to manage the visualisation of spatial, temporal and symbolic information. The following definitions are built in to the SDL type system.

```
Any = RECORD tag : STRING END;

PROCEDURE (a : Any) Handle(msg : Message) : BOOLEAN;
BEGIN
  RETURN FALSE;
END Handle;

Object = RECORD (Any) END;

Message = RECORD (Object) END;
```

Type `ANY` is a root type for all record types (user and system). Type `Object` is a root type for all user-defined record types. Objects of type `Message` are sent by the system to enquire how to handle visualisation of user types. An object defines its visualisation by overriding the `Handle` method defined in type `Any`.

The visualisation system uses *layers* to define what is displayed in a view. A layer may be simultaneously displayed in multiple views. Layers use a notification mechanism to inform views that their contents have changed. In the MVC paradigm, layers are models.

The important categories of layers are described below.

```
SimpleLayer
  SimpleObjectLayer
    HypothesisLayer
    SpatialObjectLayer
  SpatialImageLayer
```

SimpleObjectLayers encapsulate a set of objects. The contents of the object set may be managed using the `AddObject` and `RemoveObject` methods. *SpatialImageLayers* display images in a spatial display. *SpatialObjectLayers* display spatial attributes geometrically using lines and points. *HypothesisLayers* define sets of hypotheses to be displayed within the visualisation system.

Views expect to be supplied with particular kinds of layers to define their contents.

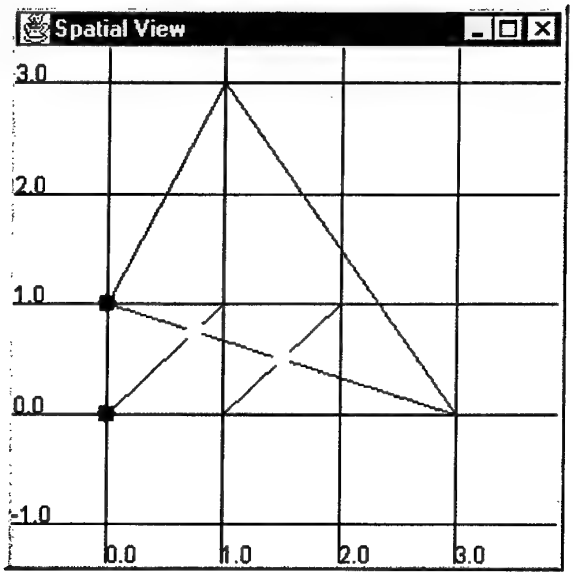

Figure 1: Sample spatial display comprising two layers.

View	Expects	Via
`SpatialView`	`SpatialLayer`	`AddLayer / RemoveLayer`
`TemporalView`	`SimpleObjectLayer`	`SetLayer`
`ObjectView`	`Object`	`SetObject`
`MainView`	`HypothesisLayer`	`SetHypotheses`

Views are exposed to SDL via restricted interfaces, rather than complete Java classes. This keeps the interfaces simple, and avoids SDL having to import definitions for classes used to implement the views (eg. javax.swing).

8.1.1 SDL.SpatialViewInterface

A spatial view may be created by one of the following:

```
view := System.NewSpatialView();
view := System.MainView().getSpatialView();
```

The first form creates an spatial view in its own window. The second form returns the spatial view component of the main view (see 8.1.3).

Figure 1 shows a sample spatial view composed of a number of layers. A layer has a user-defined identifier, and is associated with a set of objects to be included in the display. Objects within a layer define their own interpretation in terms of spatial primitives (ie. lines and points). An "active" layer is one in which the objects may be manipulated using the mouse pointer. `SpatialViewInterface` exposes the following Java functions:

```
interface SpatialViewInterface {
  public void AddLayer(SpatialLayer l);
  public void RemoveLayer(SpatialLayer l) throws RunTimeError;
  public void SetRect(double x1, double y1, double x2, double y2);
}
```

AddLayer and RemoveLayer control which layers are shown in the view. SetRect defines the bounding box for the portion of the layer to be displayed.

The following example illustrates how to construct a spatial display with two layers. The constants position and trajectory define identifiers which the system will use to refer to the layers. The example defines an inactive position layer, and an active trajectory layer using a custom colour (semi-opaque white).

```
TYPE
  ObjectLayer = JAVA SDL.SpatialObjectLayer;
  SpatialView = JAVA SDL.SpatialViewInterface;
  ImageLayer = JAVA SDL.SpatialImageLayer;
  Colour = JAVA java.awt.Color;

CONST
  position = 0;
  trajectory = 1;

VAR
  layerTrajectory, layerPosition : ObjectLayer;

BEGIN
  layerPosition := NEW ObjectLayer(position);
  layerTrajectory := NEW ObjectLayer(trajectory, TRUE,
    NEW Colour(255, 255, 255, 128));

  view := System.NewSpatialView();
  view.AddLayer(layerPosition);
  view.AddLayer(layerTrajectory);
```

The functions AddObject and RemoveObject can be used to control which objects are displayed in each layer. For example, the following adds all instances of Robot to all layers:

```
  FOREACH l IN { layerPosition, layerTrajectory} DO
    FOREACH r IN Robot DO
      l.AddObject(r)
    END
  END
```

When the state of an object changes, its representation on a layer may be updated by calling AddObject again.

The following messages are used by the system to define the representation of objects in a spatial display.

```
SpatialGet = RECORD (Message)
  layer : INTEGER;
  result : SET OF SPATIAL
END;

SpatialPut = RECORD (Message)
  layer : INTEGER;
  old, new : Spatial;
END;
```

The `SpatialGet` message is sent to an object to determine its representation on a layer. The `layer` field defines in which layer the system is requesting. The `result` field is set by the handler procedure to a set of spatial objects (eg. lines and points) that are to appear on the display. The handler returns `TRUE` to indicate that the result has been defined.

The `SpatialPut` message is sent to an object when a user manipulates its position on an active spatial layer. The `layer` field defines which layer the manipulation has occurred. The `old` field defines an element of the object's representation (ie. a member of the result set returned in a `SpatialGet` message) that has been manipulated. The `new` field defines the new value for that element.

The following example illustrates how this works. The representation of a `Robot` depends on which layer is displayed. On the `position` layer, its position (a point) is displayed. On the `trajectory` layer, its path (a line) is displayed. If the user modifies the trajectory, the `path` attribute is set to the new trajectory, and the internal state of the object is updated. Then its representation is refreshed on that layer.

```
Robot = RECORD
  path : LINE;          (* looped trajectory for robot *)
  pos : POINT;          (* current location of robot *)
END;

PROCEDURE (r : Robot) Handle(msg : Message) : BOOLEAN;
BEGIN
  WITH
  | msg : SpatialGet DO
    IF msg.layer = position THEN
      msg.result := { r.pos }
    ELSIF msg.layer = trajectory THEN
      msg.result := { r.path }
    ELSE
      RETURN FALSE
    END;
```

```
  | msg : SpatialPut DO
    IF msg.layer = trajectory THEN
      r.path := msg.new{LINE};
      r.Update();
      layerTrajectory.AddObject(r)
    END;
  END;
  RETURN TRUE
END Handle;
```

8.1.2 SDL.TemporalViewInterface

A temporal view may be created by one of the following:

```
view := System.NewTemporalView();
view := System.MainView().getTemporalView();
```

The first form creates a temporal view in its own window. The second form returns the temporal view component of the main view (see 8.1.3).

A temporal view is composed of a single of layer. `TemporalViewInterface` exposes the following Java functions:

```
interface TemporalViewInterface {
  public void SetRect(double x1, double y1, double x2, double y2);
  public void SetLayer(SimpleObjectLayer l);
}
```

`SetLayer` defines the layer to be displayed in the view. `SetRect` defines the bounding box for the portion of the layer to be displayed. In the temporal view, only `y1` and `y2` are significant. They indicate the earliest and latest times to be displayed.

Figure 2 shows a sample temporal display. In a temporal display, times are shown relative to a reference point. Each point in time appears as a line connecting two dots. This represents the earliest and latest possible times for the point compared to the reference point. An `INSTANT` appears on the display as a single point. An `INTERVAL` appears as a rectangular track, with the start point to the top left, and the end point to the bottom right. By default, the reference point is the origin (`@ 0 SECONDS`). Clicking on a point while depressing the `SHIFT` key causes the indicate point to be used as the reference point for the display.

The following example illustrates how to create events and display them in a temporal view.

```
TYPE
  ObjectLayer = JAVA SDL.SimpleObjectLayer;
```

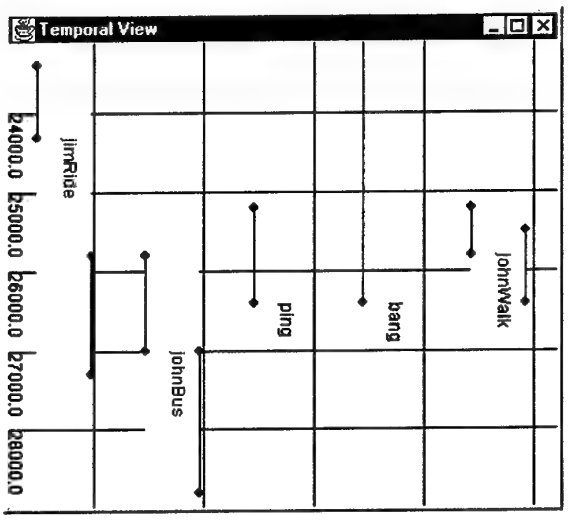

Figure 2: Sample temporal display showing three intervals and two instants.

```
  TemporalView = JAVA SDL.TemporalViewInterface;
  Interval = INTERVAL RECORD END;

VAR
  johnWalk, johnBus, jimRide : Interval;
  layer : ObjectLayer;
  view : TemporalView;

(* John leaves for work between 7:00 and 7:10. It takes him 5 to 10 minutes
to walk to the bus station. He waits between 5 to 10 minutes, and then
catches the bus to work taking 20 to 30 minutes. Jim rides his bike to
work, leaving between 6:30 and 6:45, and taking 40 to 50 minutes. *)

BEGIN
  johnWalk := NEW Interval(:tag "johnWalk");
  johnBus := NEW Interval(:tag "johnBus");
  jimRide := NEW Interval(:tag "jimRide");

  TEMPORAL
    johnWalk STARTS BETWEEN @7:00 HOURS AND @7:10 HOURS
      ALSO HAS DURATION RANGE 5 TO 10 MINUTES;
    johnBus STARTS RANGE 5 TO 10 MINUTES AFTER johnWalk ENDS
      ALSO HAS DURATION RANGE 20 TO 30 MINUTES;
    jimRide STARTS BETWEEN @6:30 HOURS AND @6:45 HOURS
      ALSO HAS DURATION RANGE 40 TO 50 MINUTES;
  END;

  layer := NEW ObjectLayer();
```

```
FOREACH i IN Interval DO layer.AddObject(i) END;

view := System.MainView().getTemporalView();
view.SetLayer(layer);
```

8.1.3 SDL.MainViewInterface

The SDL main view is a composite view for visualising temporal and spatial data, along with symbolic information. A sample display is shown in Figure 3. The display consists of the following major parts:

A *Spatial Display* (top left) shows one or more layers of spatial information (see 8.1.1).

A *Temporal Display* (top right) shows a layer of temporal information (see 8.1.2).

An *Object Display* (bottom centre) shows symbolic information about selected objects. As the user clicks to select objects from the display, the attributes of these objects are shown in the object display. Attributes referring to other objects appear as "hyper-links" which can also be selected.

A *hypothesis selector* (top centre) allows the user to be presented with a number of relevant hypotheses which they may choose to display. Each hypothesis acts as a filter on the temporal and spatial displays, restricting the display to those objects in the selected hypothesis. Alternatively, the user may select "all hypotheses" or "no hypothesis".

A *navigation tool-bar* (top centre) allows the user to step forward and back along a sequence of visited objects. It also controls the depth to which object structures are expanded in the object display.

The system normally has a single main view, which is instantiated using:

```
view := System.MainView();
```

This creates the main view if not already open. `MainViewInterface` defines the following:

```
interface MainViewInterface {
  public void SetHypotheses(HypothesisLayer layer);
  public SpatialViewInterface getSpatialView();
  public TemporalViewInterface getTemporalView();
  public ObjectViewInterface getObjectView();
  public NavigatorInterface getNavigator();
}
```

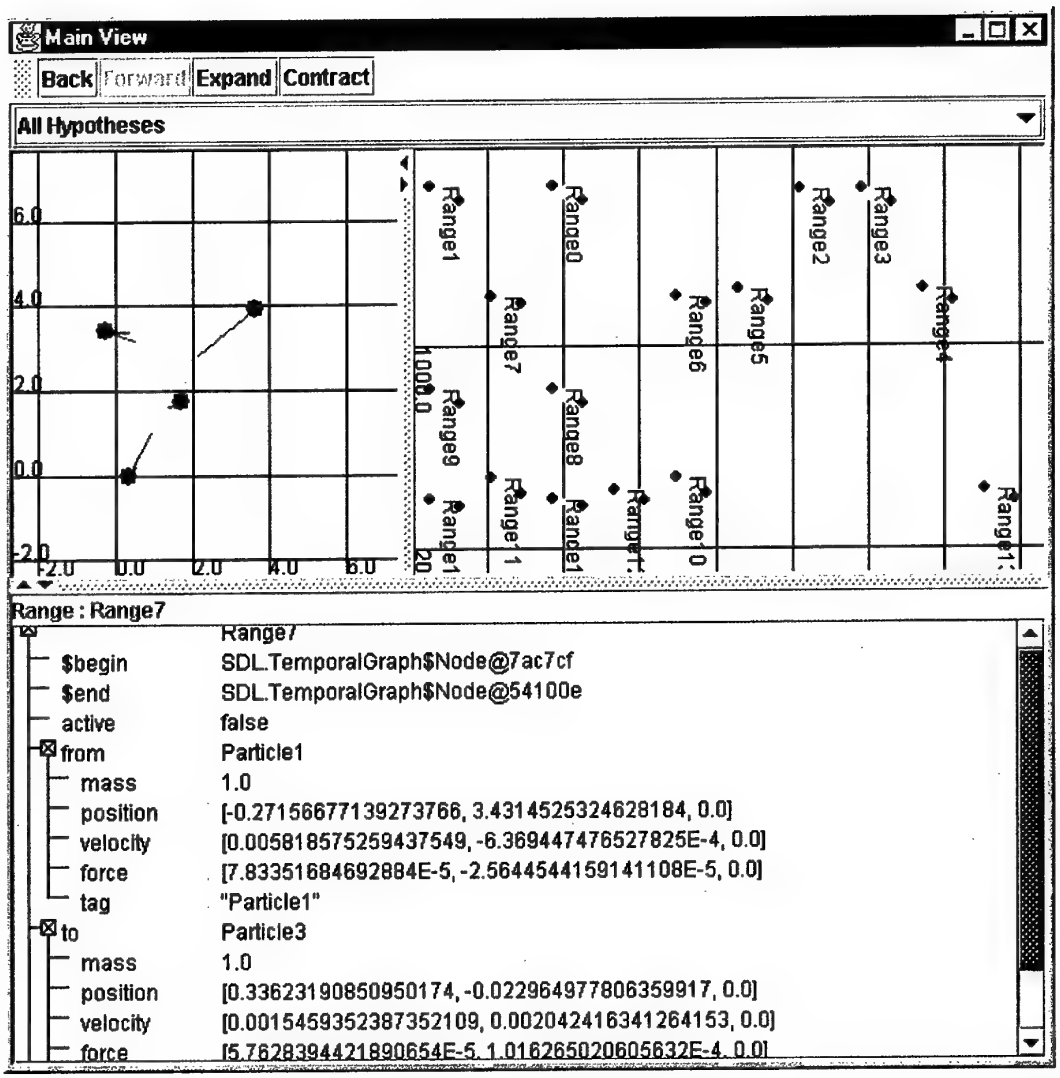

Figure 3: Sample main view showing spatial display (top left), temporal display (top right) and object display (bottom).

An object of type `JAVA SDL.HypothesisLayer` defines the hypotheses to be considered for display. In practice, there may be hypotheses that are not relevant for display, so it is not appropriate to display all hypotheses in the system. The hypothesis layer allows only those relevant to the scenario to be displayed.

The SDL system uses an internal `Hypothesis` type to represent each hypothesis. These are generated using the HYPOTHESIS statement. Hypotheses may be used in the following ways:

- A Hypothesis may be assigned to an attribute of an object. Hypothetical objects *must* be assigned a hypotheses as they are instantiated.
- A Hypothesis may be used to qualify a `HYPOTHESIS` or `TEMPORAL` statement.
- A Hypothesis may be deleted using `DELETE`. This removes the hypothesis and all associated hypothetical objects.

Within SDL, any user data associated with a hypothesis must be stored in a user-defined type. Typically, this is done as follows:

```
TYPE
  MyHypothesis = RECORD
    hypothesis : Hypothesis;
    ... user-defined data here
  END;
```

The objects defined in a hypothesis layer are *user* hypothesis objects, not system `Hypothesis` objects. The system determines the displayed name for a hypothesis using the `HypothesisGet` message:

```
HypothesisGet = RECORD (Message)
  name : STRING;
  hypothesis : Hypothesis;
END
```

These messages are sent to all objects in the view's hypothesis layer. The user must return a system `Hypothesis` object, and a displayed name for the hypothesis. For example:

```
PROCEDURE (m : MyHypothesis) Handle (msg : Message) : BOOLEAN;
BEGIN
  WITH msg : HypothesisGet DO
    msg.name := m.tag;
    msg.hypothesis := m.hypothesis;
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END;
END Handle;
```

Note that user hypotheses may also have spatial and temporal representations. In this case, Handle would respond to these messages as well.

```
TYPE
  HypothesisLayer = JAVA SDL.HypothesisLayer;
  ...

VAR
  layerHypotheses : HypothesisLayer;
  ...

  (* Create Hypothesis layer for Main View *)
  layerHypotheses := NEW HypothesisLayer();
  System.MainView().SetHypotheses(layerHypotheses);
  ...

  (* Create user object corresponding to system hypothesis *)
  HYPOTHESIS h
    m := MyHypothesis(:hypothesis h, ...);
    ...
    (* Add the new hypothesis to the display. *)
    layerHypothesis.AddObject(m);
  END;
  ...
```

8.1.4 SDL.NavigatorInterface

A SDL.NavigatorInterface defines how the system handles objects and hypothesis in the main view. These interfaces are created by the system.

```
interface NavigatorInterface {
  static final int ShowMatch = 0;
  static final int ShowAll = 1;
  static final int ShowNone = 2;

  boolean Link(Object target);
  boolean Included(Object target);
  void SetMode(int mode, Hypothesis hypothesis);
}
```

The procedure SetMode defines how the system handles displayed hypotheses. If mode=ShowAll, all hypotheses in the hypothesis layer will be displayed. If mode=ShowNone, no hypotheses will be displayed. If mode=ShowMatch, the hypothesis corresponding to hypothesis will be selected for display.

The procedure Link causes the object view to display the the target object. The object is placed in the navigation list and is accessible via forward and back commands.

The procedure `Included` returns `TRUE` if and only if the `target` object is displayable with the current settings.

The `SDL.NavigatorInterface` functions are intended for use by the system, but may be also called *with care* by the user.

8.1.5 SDL.ObjectViewInterface

The `SDL.ObjectViewInterface` defines the interface to the object viewer. This viewer displays the values of attributes of objects. The user defines the displayed object by clicking on objects in the spatial and temporal displays, or clicking on references from other objects in the object display. The user may navigate back and forward using web-browser like navigation controls. It is also possible to control the depth to which object references are expanded in the object display using on-screen controls.

```
interface ObjectViewInterface {
  public void SetObject(Object o, int level);
}
```

The procedure `SetObject` causes the object view to display object `o`, expanding object references to a depth of `level`.

9 Notification

SDL includes a notification mechanism to manage interaction with other software systems. A client system may register interest in the following events:

- Creation of a new instance of a record type.
- Deletion of an instance of a record type.
- Changes to particular attributes of a record.

Registration is achieved using request commands (see 9.2). Each command consists of a predicate where the command arguments are enclosed in parentheses. Commands are parsed and dispatched by SDL using a `SDL.NotifyParser`, which implements the `SDL.ForeignQuery` interface:

```
interface ForeignQuery {
  public String Query(String query);
}
```

Commands are passed to the `ForeignQuery` handler as strings, and results are returned as strings. Normally, the result of a query is either `'Success'()`, or `'Error'('reason')`.

The notification process normally works as follows:

- After reaching a significant point in its deliberation, the SDL process calls `System.Notify` to inform the client process of the current state of the data base.
- The SDL process generates notification messages which are forwarded to the client.
- The SDL process services queries from the client. Typically, having been notified about a new object, the client will want to register notifications for particular attributes, or to request the current values of attributes. The client may also call SDL procedures to register changes in its own state (eg. its intentions) which may affect the situation assessment.
- When the SDL process receives the `'done'()` command, it returns from `System.Notify` and deliberation resumes again.

Internally, the system implements foreign notification handlers using the following interface:

```
interface ForeignNotify {
  public void Notify(String message);
  public void Answer(ForeignQuery query);
}
```

The `Notify` procedure is used to send notifications to the client. The `Answer` procedure is used to handle queries from the client. Two implementations of `ForeignNotify` are available. `SDL.TestNotify` implements a dummy notifier that simply outputs notification messages to `System.out`. `SDL.DMarsNotify` implements a notification agent within the DMars system using a JNI interface to the vendor's messaging API. This is currently experimental and has not been completely implemented.

9.1 Representation of SDL values

The notification system has been developed primarily to interface to DMars. A requirement for such systems is to transform the hierarchical SDL data structure into a set of relations or predicates. SDL employs object "tags" to indicate a reference to an object. A tag uniquely identifies an object within the SDL system. Tags are generated automatically by the system, or may be assigned by the user.

Table 1 outlines the predicate representations for SDL values. Scalar values (INTEGER, REAL, STRING, BOOLEAN) are represented using the `'Base'` predicate. The value `NIL` is represented by the `'Nil'` predicate. Sets and sequences have their element values enumerated within an `'Array'` predicate. Potential values use the `'Potential'` predicate to enclose a list of `'Element'` predicates, one for each association between value and certainty.

Records are represented in two ways. The long form uses the `'Record'` predicate to enclose a list of `'Attribute'` predicates. Each `'Attribute'` specifies the attribute name, and the value for the named attribute. The short form uses just the `'Object'` predicate to specify the tag for the object. Normally, the outermost record is mapped to the long form, with any embedded references to records using the short form.

SDL Value	Predicate mapping
`NIL`	`'Nil'()`
`TRUE`	`'Base'(true)`
`42`	`'Base'(42)`
`4.2`	`'Base'(4.2)`
`"A String"`	`'Base'('A String')`
`{1, 2, 3}`	`'Array'('Base'(3),'Base'(2),'Base'(1))`
`[1, 2, 3]`	`'Array'('Base'(1),'Base'(2),'Base'(3))`
`{{1 CF 0.1, 2 CF 0.2}}`	`'Potential'('Element'('Base'(2),'Base'(0.2)),` `'Element'('Base'(1),'Base'(0.1)))`
`NEW A(:i 1, :tag "A22")`	`'Record('Attribute'('i','Base'(1)),` `'Attribute'('tag','Base'('A22')))`
`NEW A(:i 1, :tag "A22")`	`'Object('A22')`

Table 1: Mappings for SDL values

9.2 Notification Requests

Notification requests have the syntax specified in Table 2. For most queries, the result is either `'Success'()` or `'Error'('reason')`, where `'reason'` is an explanation of the error context.

The `RequestNew` command requests notification for new instances of the named class. For example:

```
'RequestNew'('Notify.A')
```

will cause new instances of the class `Notify.A` to be reported. Notifications have the form:

```
'NotifyNew'(RecordType,Recordname)
'NotifyDelete'(RecordType,Recordname)
```

For example, the message:

```
'NotifyNew'('A','A22')
```

means that there is a new instance of class `A` with tag `A22`[3].

A number of commands exist to request information about attributes of objects. These employ the `RecordSpec` construct, which names the attributes of an object. In its simplest form, a `RecordSpec` is simply an object tag, followed by a list of attribute names separated by commas. For example:

[3]Note that SDL does not currently report the module in which a type is defined, so there is a potential ambiguity if types of the same name occur in different modules

```
Integer = [ "-" ] Digit { Digit } .
Real = Integer "." {Digit} [ "E" Integer ] .
String = Letter { Letter | Digit } .

Quote = "'" .
QuotedString = Quote String Quote .

QualIdent = Quote String "." String Quote .

ClassName = QualIdent .

RecordName = QuotedString .

AttributeSpec = Quote String { "." String } [ ":" String ] Quote .

RecordSpec = RecordName { "," AttributeSpec } .

ProcName = QualIdent .

Value = "'" String "'" | Integer | Real .
CallArguments = ProcName { "," [":" String ] Value } .

Command =
    "'RequestNew'" "(" ClassName ")"
  | "'RequestCurrent'" "(" RecordSpec ")"
  | "'RequestChangeSet'" "(" RecordSpec ")"
  | "'RequestChangeRemove'" "(" RecordSpec ")"
  | "'Call'" "(" CallArguments ")"
.
```

Table 2: Syntax of Notification requests

```
'A22','i','j'
```

means attributes i and j of the object with tag A22.

SDL allows a collection of objects to be viewed as an aggregate object. This helps to avoid the number of object references that must be exchanged between SDL and the client. In the AttributeSpec definition above, an arbitrary number of indirections may be introduced together with an attribute alias.

```
TYPE
  B = RECORD
    i : INTEGER;
  END;

  A = RECORD
    i : INTEGER;
    b : B;
  END;
```

In the above type definition, field b of type A refers to an object of type B. If A22 is the tag of an object of type A, then the RecordSpec:

```
'A22','b.i:bi'
```

means attribute i of the object referenced by attribute b of the object with tag A22, referred to by the alias bi.

The command 'RequestCurrent' generates an immediate notification of the specified attributes of an object. Note that the attribute values are returned in a notification message, not in the result of the request command. The general form for attribute notifications is:

```
'AttributeValue'(RecordType,RecordTag,AttributeName,Value)
```

where RecordType is the name of the record's type, RecordTag is the record's tag, AttributeName is the name or alias of the attribute, and Value is the representation of the value of the attribute. For example:

```
'AttributeValue'('A','a','i','Base'(2))
'AttributeValue'('A','a','bi','Base'(3))
```

The command 'RequestChangeSet' requests notifications of changes to the specified attributes of an object. The command 'RequestChangeRemove' cancels notifications of changes to the specified attributes. Example:

```
'RequestChangeSet'('a','i','b.i:bi')
```

In this example, the client requests notification for attribute `i` of object `a`, and attribute `i` of the object referenced by attribute `b` of object `a` using the alias `bi`. This means that changes to the referenced object are treated as changes to the referring object.

The command `'Call'` calls a procedure in a SDL module, passing a list of arguments. Currently, only STRING, INTEGER, and REAL argument types are handled by the notification parser. The argument list must match the formal parameter types specified in the SDL procedure definition.

Unlike other commands, the `'Call'` command returns a result when successful. For example,

```
'Success'('Base'(1))
```

indicates that the procedure call returned the value `1`. The result `'Success'('Nil'())` means *either* the procedure returned the value `NIL`, or the procedure did not return a value.

10 Example-Submarine Situation Assessment

10.1 Overview

This work was performed to gain a better understanding of what takes place during a submarine situation assessment. Submarine situation assessment is a process that submariners perform to assess the current situation. Situation assessment is done by a team of submariners in loose hierarchical fashion. The final authority rests with the commander. The focus of the current system is the assessment performed by the commander. The task is complicated by the need to remain covert. The strength of the submarine has always been its element of suprise. To achieve this, the submarine's primary sensors are the passive sonar which listens to sound emitted by other vessels, ESM (Electronic Support Measures) which monitors electro-magnetic emission, and the periscope which allows an operator to see other vessels. Being passive sensors they are especially prone to environmental effects, and conditions of the emitting source. This greatly increases uncertainty in the gathered information.

Re-association occurs when a lost contact is associated with a new contact. A contact occurs when a sensor indicates the presence of an entity (natural or man-made). Re-association is a process prone to error. A contact becomes lost when none of the sensors on board can detect it. Multiple hypotheses are used to allow the operator to re-associate a new contact with a lost contact. The hypothesis with the lost contact is "clone". In the clone hypothesis the free contact is re-associated with the new contact. In the original hypothesis a new entity of unknown classification is created and is associated with the new contact. Each hypothesis is allowed to run until new evidence eliminates the hypothesis. The deductions about the entities in a hypothesis can take place within a hypothesis. This provides an interpretation of the situation based on a particular hypothesis.

10.2 Knowledge base

Information about opponent capabilities and equipment help in identifying a vessel, and deciding the threat a system poses. Information about opponents was encoded using an object-oriented knowledge structure as shown in Figure 4. The hierarchical knowledge base is broken up into three main branches. Two branches encompass the sub-systems (weapons and sensors) that can be on board a vessel. The remaining branch is a decomposition of various types of vessel. Vessels will contain sets of sub-systems. This structure allows us to trace which vessel contains a specific sub-system. Finding out which systems exist on a particular entity will tell us the type of vessel the entity is. This information is static in nature.

```
PROCEDURE possibleVeh(ss: SENSOR): SET OF VEHICLE;
VAR
vehList : SET OF VEHICLE;
BEGIN
     vehList := FROM veh: VEHICLE
           WHERE ss IN veh.Sensor
   SELECT veh  END;

     RETURN vehList;
END possibleVeh;
```

Other information is temporal in nature. Sensory information is always temporal because the information is only valid at the time the information was extracted from the sensor. Action performed by the submarine is also temporal in nature. For instance, the fact that a submarine is making a course change is only valid at the time the action is being taken. The structure for these items information is shown in Figure 5. Note that sensory readings are declared as "PERSISTENT". That is, the event object is maintained in the data base but marked "inactive" which means that it no longer matches patterns in rules. History of the event remains for matching temporal constraint.

10.3 Rule base

10.3.1 Reasoning with contacts

These rules deal with making assessments using given sensory information without assuming any specific classification. They make use of temporal and non-temporal data. An example such a rule is :

```
RULE MovingLeft
EVENT
  MOVINGLEFT {contact <C>  }
WHEN
  BEARINGRATE cut1 { contact <C> active TRUE rate <rate1> : rate1 < -0.015 } &
  ~ BEARINGRATE cut2 { contact <C> active FALSE rate <rate2>
       : cut1#cut2: rate2 > -0.015 :TEMPORAL cut2 HAPPENS WITHIN 10 MINUTES
       OF cut1 HAPPENS; END}
```

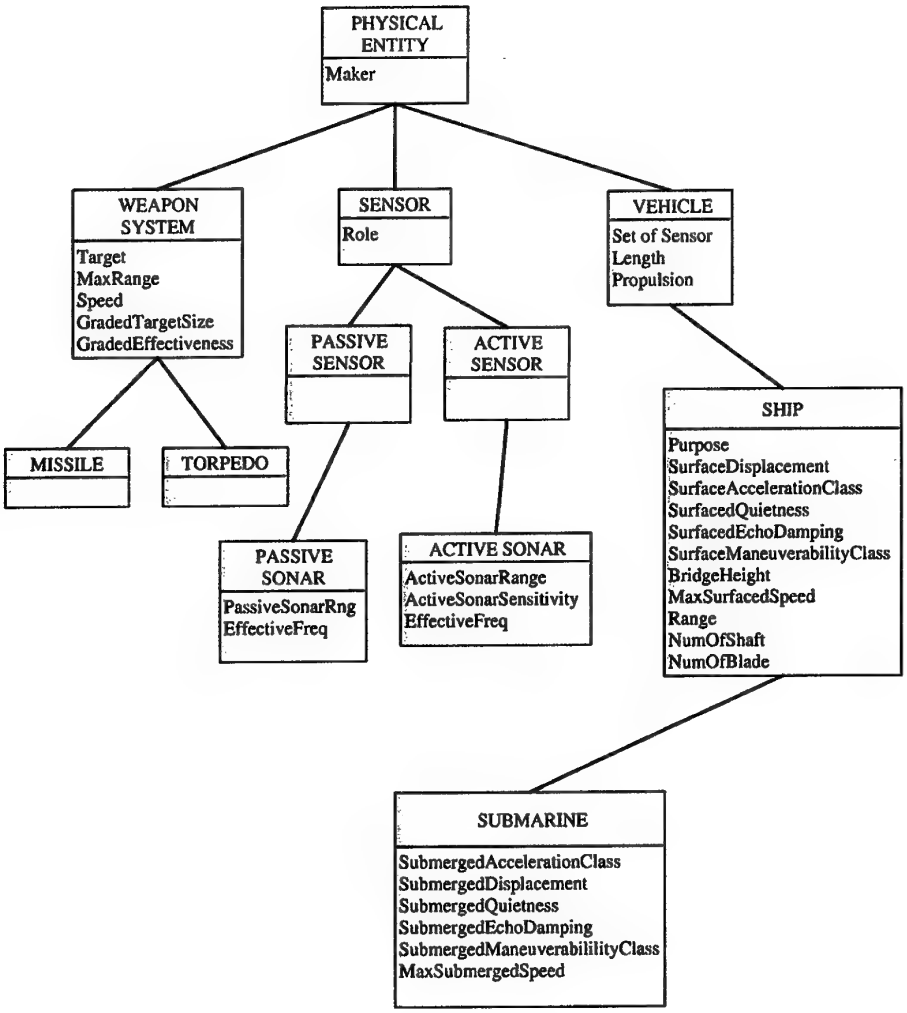

Figure 4: Structure for storing background information about vessels of interest.

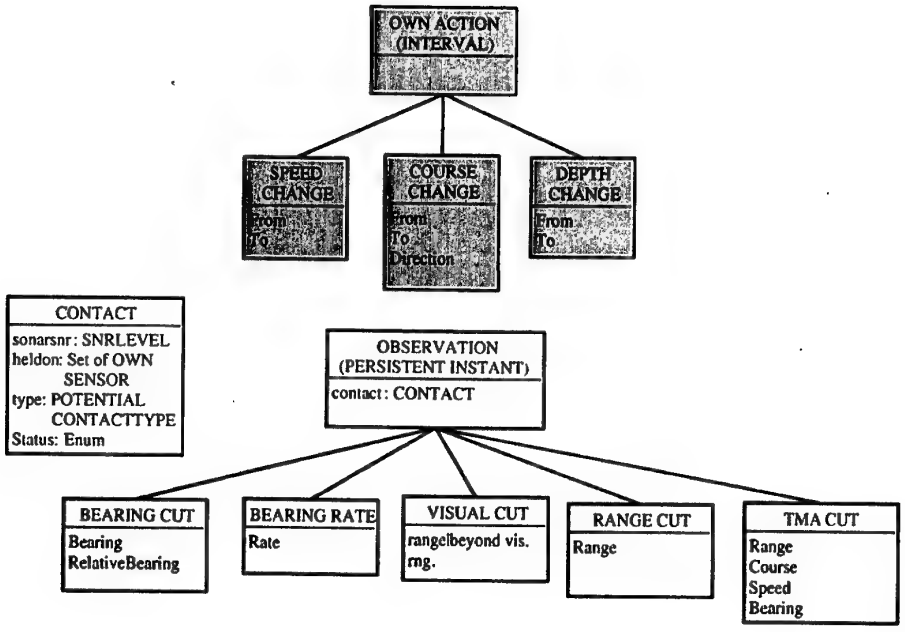

Figure 5: Structure for storing sensory information.

```
ACTIVE
  STARTS WITHIN 10 MINUTES OF cut1 HAPPENS ALSO STARTS BEFORE cut1 HAPPENS
INACTIVE
  ENDS WITHIN 10 MINUTES OF cut1 HAPPENS ALSO ENDS BEFORE cut1 HAPPENS
END MovingLeft;
```

This rule records the period in which bearing rate (rate and direction the bearing was changing) of the contact was noticably moving towards the left hand side. It is an event rule that creates a "MOVINGLEFT" instance when the bearing rate indicates the contact is noticably moving towards the left, and no contrary evidence is present in the last 10 minutes. The instance is set to have started 10 minute earlier. This rule remains active until a bearing rate change to not left. The instance ceases to be true sometime within 10 minutes prior to the contrary evidence.

10.3.2 Reasoning within hypothesis

Rules and procedures primarily exist for generating conclusions made about entities. They access information about the entities, their associated contact, and background knowledge. Any conclusions they make are embedded within the hypothesis. Take for instance this rule:

```
RULE FROMRANGEOFDAY
IF  ENTITY e {hypothesis <H> classification <class> curContact<trk> } &
    Self.CONTACT trk {sensor <<ss>>} &
    Self.OWNSONAR ss{standardrange <srng1>} &
```

```
    Self.BEARINGCUT brg {contact<trk> bearing <angle>} &
    Self.RANGEOFTHEDAY r1 {sensor <ss> type <class> range <maxrng>} &
    ~ (Self.TRACK trk {heldon <<ss2>> :ss2 IS Self.OWNSONAR :ss2 # ss} &
       Self.RANGEOFTHEDAY {sensor <ss2> type <class> range <rng1> :rng1 < maxrng} )&
    Self.OWNSENSOR ss3 {} &
    Self.OWNSONAR ss3 {coverage <<sector>> standardrange<srng2> :srng1 # srng2
                        :sector.enclose(angle) :NOT (ss3 IN trk.sensor)} &
    Self.RANGEOFTHEDAY {sensor <ss3> type <class> minrange <minrng>} &
    ~ (Self.OWNSENSOR ss4 {} &
       Self.OWNSONAR ss4 {coverage <<sector>> standardrange<srng3> :srng1 # srng3
                        :sector.enclose(angle) :NOT (ss4 IN trk.sensor)} &
       Self.RANGEOFTHEDAY {sensor <ss4> type <class> minrange <rng2>:rng2 > minrng})&
    ~ SENSORRANGE {hypothesis <H> source<e> min <minrng> max <maxrng>}
THEN
    TEMPORAL NEW SENSORRANGE (:hypothesis H, :entity e, :min minrng, :max maxrng,
                :difference (maxrng - minrng)) HAPPENS AT brg HAPPENS; END;
END FROMRANGEOFDAY1;
```

It accesses information about the entity classification, the sensors the contact was held on, and background information about sonar capabilities of own submarine. Using that, it deduces an expected band of ranges the entity should be within. The conclusion is recorded in a hypothetical record placed in the same hypothesis as the referenced entity.

10.4 Creating and terminating hypothesis

Hypotheses are created and destroyed as we see fit. This is achieved by a set of rules and procedures to govern the creation and destruction of hypotheses. One way hypotheses are created is when a free entity is re-associated with a new contact. The procedure below shows how this is done.

```
PROCEDURE CreateEntity( c: Self.CONTACT);
VAR
    h : MYHYPOTHESIS;
    elist : SET OF ENTITY;
    entity : ENTITY;
    likelyclass : SET OF NavalKB.CONTACTTYPE;
    class : NavalKB.CONTACTTYPE;
    detail : STRING;
BEGIN

    (* Find all free entities of all hypotheses *)
    elist := FROM e :ENTITY WHERE e.status = FREE SELECT e END;
    FOREACH entity IN elist DO
        IF passConditions( entity, c) THEN
            (* Duplicate hypothesis of entity *)
            HYPOTHESIS hclone FROM entity.hypothesis
                h := NEW MYHYPOTHESIS(:h hclone, :del FALSE );
                (* Associate free entity in duplicate hypothesis to new contact. *)
                UPDATE
                    MAP(hclone, entity).originalContact := entity.curContact;
                    MAP(hclone, entity).curContact := c;
                    MAP(hclone, entity).status := REASSOCIATED;
                END;

            END;
        END;
    END;

    (* Create an unknown entity for each hypothesis without one associated with contact *)
    FOREACH h IN MYHYPOTHESIS DO
```

```
        IF h.del = FALSE THEN
            elist := FROM e :ENTITY WHERE e.hypothesis = h.h, e.curContact = c SELECT e END;
            IF elist = {} THEN
                (* Create an unknown entity. *)
                  entity := NEW ENTITY (:hypothesis h.h, :originalContact c, :curContact c,
                      :range UNKNOWN, :threat 3, :classification NavalKB.UNKNOWN
                      , :class NEW CLASSVARIANT (:unknown), :status CREATED,
                      :path [], :lpath {}, :pospath {});
            END;
        END;
      END;
END CreateEntity;
```

The destruction of these hypotheses is governed by a set of constraints. When a hypothesis violates one of these constraints, it is set as invalid and can be eliminated. These rules play the very important role of containing the number of hypotheses. Three such constraints have been established. They are:

1. Type mismatch: Mismatch between entity type and dominant type of the associated track. For instance, where a submarine entity was re-associated with a contact that was later classified as a warship.

2. Spatial improbability: New information indicates the likely position of the entity, at time t2. We also have the likely position of the entity prior to being lost, at time t1. This rule states the entity in question must be able to reach the new position from the old position in the time between t1 and t2.

3. Multiple submarines: The risk of friendly fire and collision is very high if two submarines are deployed in close proximity. Thus, the likelihood of two submarines being detected in close proximity is remote. A hypothesis where two submarine entities exist in close proximity is invalid.

An example of one of these constraints is shown below.

```
RULE KillBadAssociation
IF ENTITY e {hypothesis <H> classification <class> curContact <trk>
          status REASSOCIATED} &
   Self.CONTACT trk {type <<etype>> :NOT (NavalKB.UNKNOWN IN
          LIKELY(trk.type)):NOT (class IN LIKELY(trk.type))} &
   MYHYPOTHESIS myh {h <H>}
THEN
   PRINTLN "Bad association.";
   UPDATE
       DELETE myh;
       DELETE H;
   END;
END KillBadAssociation;
```

Note the procedure always creates a new unknown entity in each of the original hypotheses, since the new contact could potentially be a new entity. Classification information arriving determines the entity classification. The classification can change again as better information comes in. It is desirable to use the new classification information instead of waiting to make sure the contact classification will not change again. To allow this, a hypothesis is formed when the associated contact dominant classification changes. The rules governing creation and destruction of these hypotheses are shown below.

```
RULE SplitUnknown
IF  ENTITY e {hypothesis <H> classification <class> status <state1>
              curContact <trk> :state1 = CREATED: class = NavalKB.UNKNOWN} &
    Self.CONTACT trk { type <<etype>> :etype IN LIKELY(trk.type)
          :NOT (NavalKB.UNKNOWN IN LIKELY(trk.type)): NOT( etype IN e.morphTo )}
THEN
    classificationSplit( e, etype); (* create new hypothesis *)
    e.morphTo  := e.morphTo  + {etype};
END SplitUnknown;

RULE KillInvalidClassEntity
IF  Self.CONTACT trk { status <state> : (state = Self.CEASE) OR (state = Self.LOST)} &
    ENTITY e {hypothesis <H> classification <class>  curContact <trk> originalContact <trk>
              classification <class> : NOT (class IN LIKELY (trk.type) )} &
    MYHYPOTHESIS myh {h <H>}
THEN
    UPDATE
        myh.del := TRUE;
        lhypot.RemoveObject(myh);
        lspatial.UpdateAll();
        DELETE H;
        DELETE myh;
    END;
END KillInvalidClassEntity;
```

The first rule creates a new hypothesis with an entity equal to the contact's current dominant classification. The second rule deletes the hypotheses where the entity classification does not match the contact dominant classification when the contact is lost. At this point, no new information can arrive to alter the contact dominant classification.

10.5 Entering data into SAP

A set of procedures exist to form the interface to the SAP for this application. Information is passed as arguments to these procedures. Calls to these procedures could be from a simulation, or in our case from a script file containing an ordered sequence of these procedure calls. The data for our script file was extracted from a submarine on submarine exercise.

```
PROCEDURE  newRangeCut( time:INTEGER; tag: STRING; rng:REAL);
VAR
    cnct : CONTACT;
    rangeset : SET OF RANGECUT;
    range: RANGECUT;

BEGIN
    FOREACH cnct IN CONTACT DO
        (* Find contact with given tag *)
        IF cnct.tag = tag THEN
         (* Set previous RANGECUT for this contact to be inactive. *)
            rangeset := FROM t:RANGECUT WHERE t.active = TRUE
                                        , t.source = cnct SELECT t END;
            FOREACH range IN rangeset DO range.active := FALSE; END;
            (* Create new RANGECUT for this track. *)
            TEMPORAL NEW RANGECUT (:source cnct, :range rng, :active TRUE )
                                HAPPENS AT @time SECONDS; END;
        END; (* IF *)
    END; (* FOREACH *)
END newRangeCut;
```

10.6 Situation snapshot

The set of graphical snapshots of the Submarine Situation Assessment application illustrates how the multiple hypotheses work in practice. The first snapshot in Figure 6 shows only one hypothesis containing two free entities. Using previous information it was able to give an indication of the whereabouts of these two entities. One of the entities is a warship and the other was a submarine. The next three snapshots are the resultant hypotheses when a new contact was detected. The first interpretation of the situation, shown in Figure 7, is that the new contact is the submarine previously lost. The second interpretation, as shown in Figure 8, is that the new contact is the warship. The final interpretation, as shown in Figure 9, is that the new contact is a previously undetected entity. The final snapshot was taken when the new contact was later classified as a submarine, shown in Figure 10. The result is that the hypothesis where the new contact was the warship was eliminated. The hypothesis that the new contact is a new unknown entity will be eliminated based on the rule *KillInvalidClassEntity* described in previous page.

11 Conclusion

SDL is a Situation Description Language intended for use in situation assessment problems. SDL provides knowledge modelling and inference facilities for reasoning with information.

SDL is a strongly typed language, like most imperative programming languages but unlike many "artificial intelligence" languages. SDL programs are strictly checked during a compilation phase which enables many semantic errors (such as type mismatch) to be easily identified. SDL provides a rich declarative knowledge model together with pattern-based forward-chaining inference.

The details presented here are important for users and implementors of the SDL system. The examples demonstrate the potential usefulness of the system in complex tasks such as submarine situation assessment.

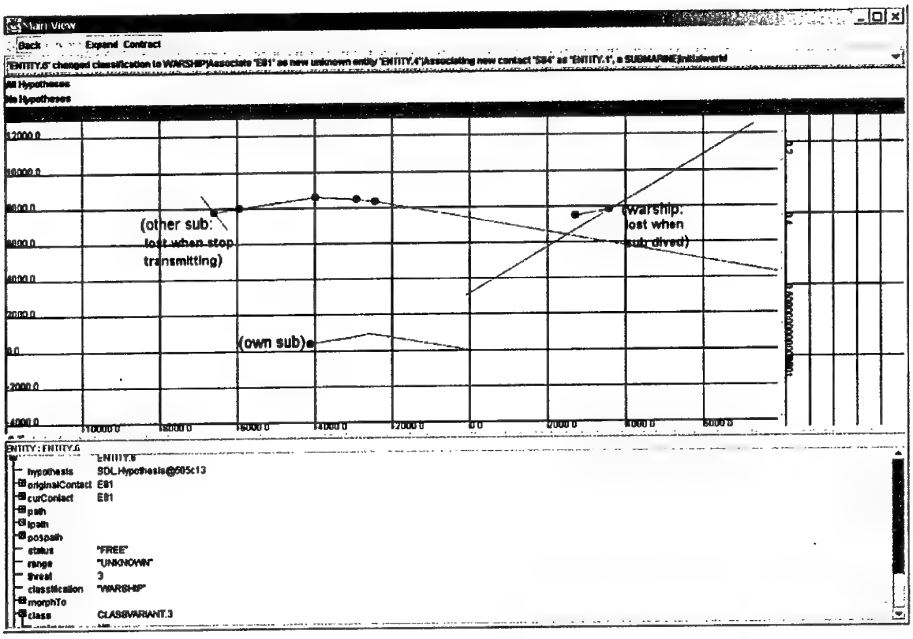

Figure 6: Spatial disposition of entity with known position.

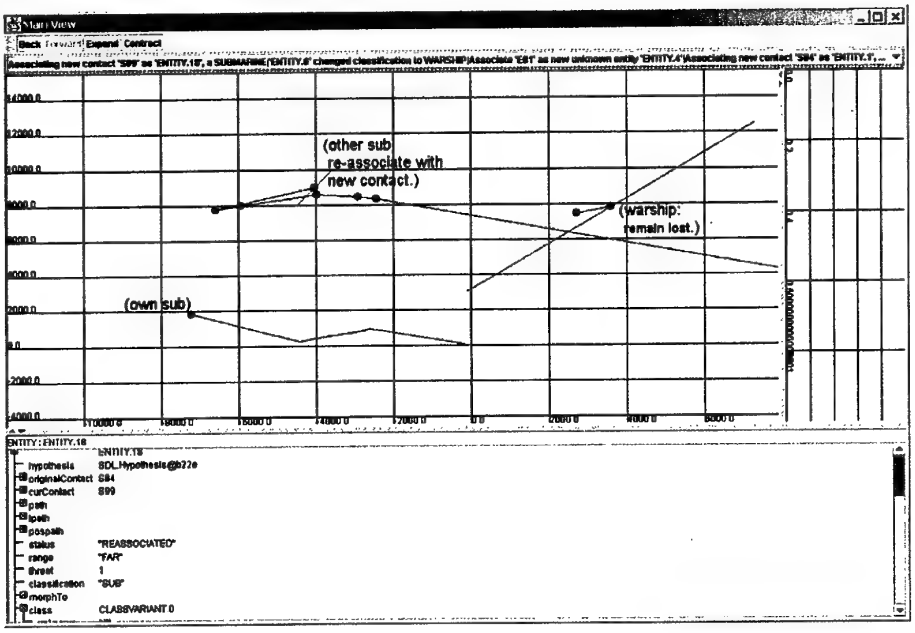

Figure 7: Associate new contact as lost submarine.

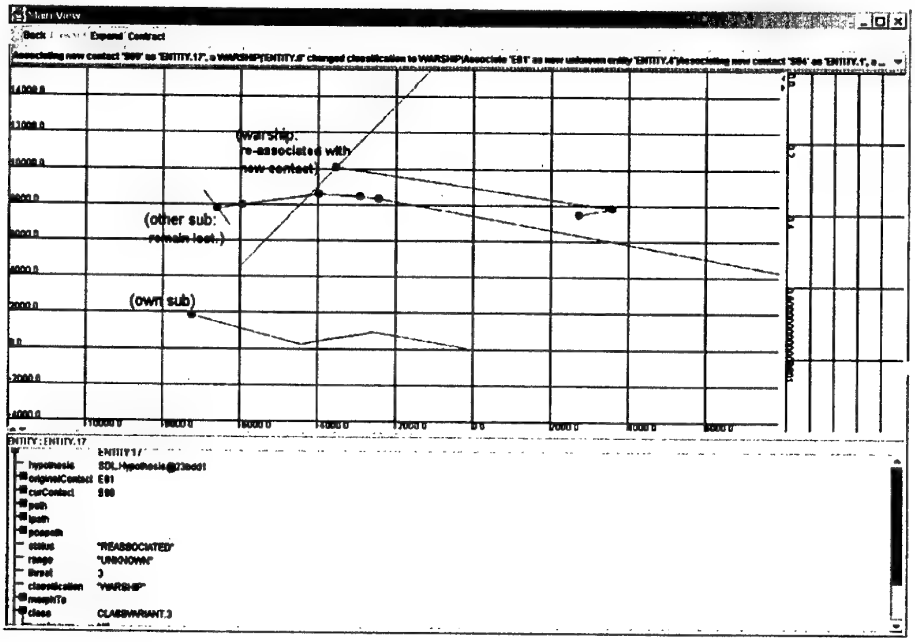

Figure 8: Associate new track as lost warship.

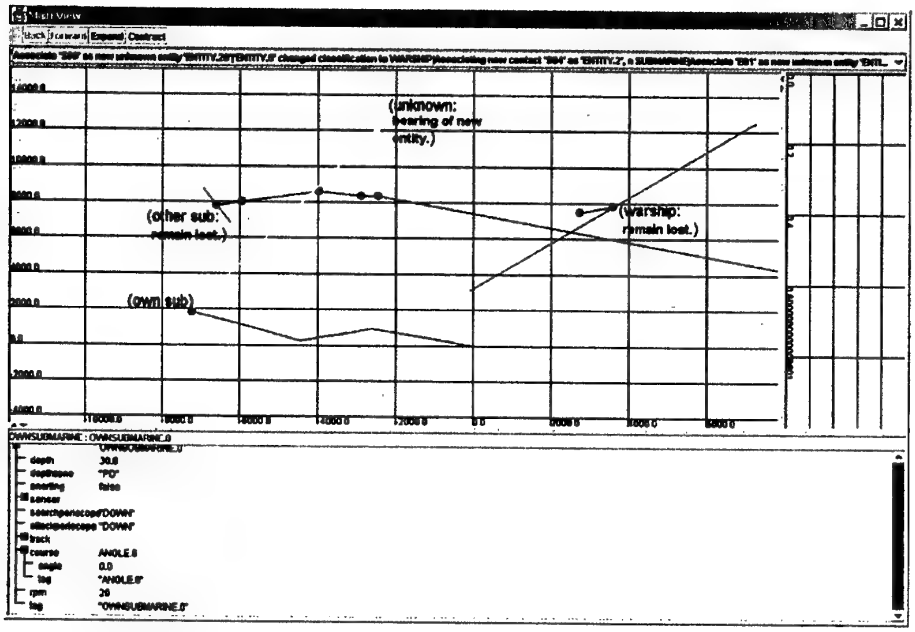

Figure 9: Associate new track as new unknown entity.

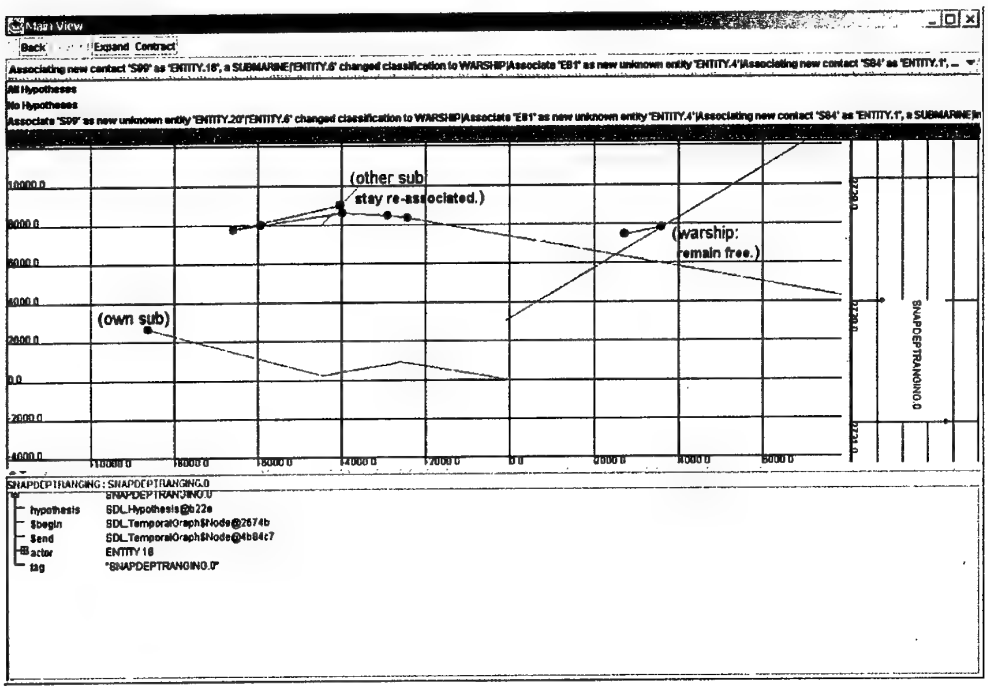

Figure 10: New information remove warship as viable hypothesis.

References

1. *CLIPS expert system shell* (n.d.) http://www.ghg.net/clips/CLIPS.html.

2. Dechter, R., Meiri, I. & Pearl, J. (1991) Temporal constraint networks, *Artificial Intelligence* **52**, 61–95.

3. Forgy, C. L. (1982) RETE: A fast algorithm for the many pattern / many object pattern match problem, *Artificial Intelligence* **19**, 17–37.

4. Greenhill, S., Venkatesh, S., Pearce, A. & Ly, T. (2002) *Representations and Processes in Decision Modelling*, Technical Report DSTO–GD–0318, DSTO, Melbourne, Australia.

5. Greenhill, S., Venkatesh, S., Pearce, A. & Ly, T. (2002) *Situation Description Language*, Technical Report DSTO–GD–0342, DSTO, Melbourne, Australia.

6. *JESS The rule engine for the Java platform* (n.d.) http://herzberg.ca.sandia.gov/jess/.

7. Lee, H. S. & Schor, M. I. (1992) Match algorithms for generalized Rete networks, *Artificial Intelligence* **54**, 249–274.

DISTRIBUTION LIST

Situation Description Language Implementation

S. Greenhill and S.Venkatesh and A. Pearce and T.C. Ly

	Number of Copies
DEFENCE ORGANISATION	
S&T Program	
Chief Defence Scientist FAS Science Policy AS Science Corporate Management Director General Science Policy Development	1
Counsellor, Defence Science, London	Doc Data Sht
Counsellor, Defence Science, Washington	Doc Data Sht
Scientific Adviser to MRDC, Thailand	Doc Data Sht
Scientific Adviser Joint	1
Navy Scientific Adviser	Doc Data Sht
Scientific Adviser, Army	Doc Data Sht
Air Force Scientific Adviser	1
Director Trials	1
Information Sciences Laboratory	
Don Perugini, C2D, Edinburgh	1
Poh Lian Choong, C2D, Edinburgh	1
Systems Sciences Laboratory	
Chief of Maritime Operation Division	1
Research Leader Combat Information Systems	1
Head Submarine Combat System	1
John Best, MOD, Edinburgh	1
Thanh Chi Ly, MOD, HMAS Stirling	1
Chris Davis, MOD, HMAS Stirling	1
Simon Goss, AOD, Fishermans Bend	1
DSTO Library and Archives	
Library, Stirling	1
Library, Edinburgh	1
Australian Archives	1
Capability Systems Staff	
Director General Maritime Development	Doc Data Sht

Knowledge Staff

Director General Command, Control, Communications and Computers (DGC4)	Doc Data Sht

Army

ABCA National Standardisation Officer, Land Warfare Development Sector, Puckapunyal	4

Intelligence Program

DGSTA, Defence Intelligence Organisation	1
Manager, Information Centre, Defence Intelligence Organisation	1

Defence Libraries

Library Manager, DLS-Canberra	1
Library Manager, DLS-Sydney West	Doc Data Sht

UNIVERSITIES AND COLLEGES

Australian Defence Force Academy Library	1
Hargrave Library, Monash University	Doc Data Sht
Librarian, Flinders University	1
Curtin Library, Curtin University	1

School of Computing, Curtin University

Stewart Greenhill	1
Svetha Venkatesh	1

University of Melbourne

Adrian Pearce	1

OTHER ORGANISATIONS

National Library of Australia	1
NASA (Canberra)	1
AusInfo	1

INTERNATIONAL DEFENCE INFORMATION CENTRES

US Defense Technical Information Center	2
UK Defence Research Information Centre	2
Canada Defence Scientific Information Service	1
NZ Defence Information Centre	1

ABSTRACTING AND INFORMATION ORGANISATIONS

Library, Chemical Abstracts Reference Service	1

Engineering Societies Library, US	1
Materials Information, Cambridge Scientific Abstracts, US	1
Documents Librarian, The Center for Research Libraries, US	1

INFORMATION EXCHANGE AGREEMENT PARTNERS

Acquisitions Unit, Science Reference and Information Service, UK	1
Library – Exchange Desk, National Institute of Standards and Technology, US	1

SPARES

DSTO Edinburgh Library	5
Total number of copies:	49

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA				1. CAVEAT/PRIVACY MARKING
2. TITLE Situation Description Language Implementation			3. SECURITY CLASSIFICATION Document (U) Title (U) Abstract (U)	
4. AUTHORS S. Greenhill and S.Venkatesh and A. Pearce and T.C. Ly			5. CORPORATE AUTHOR Systems Sciences Laboratory PO Box 1500 Edinburgh, South Australia, Australia 5111	
6a. DSTO NUMBER DSTO–GD–0342	6b. AR NUMBER 012-486		6c. TYPE OF REPORT General Document	7. DOCUMENT DATE November, 2002
8. FILE NUMBER M9505/23/30	9. TASK NUMBER LRR 98/081	10. SPONSOR	11. No OF PAGES 39	12. No OF REFS 7
13. URL OF ELECTRONIC VERSION http://www.dsto.defence.gov.au/corporate/reports/DSTO–GD–0342.pdf			14. RELEASE AUTHORITY Chief, Maritime Operations Division	
15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT *Approved For Public Release* OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111				
16. DELIBERATE ANNOUNCEMENT No Limitations				
17. CITATION IN OTHER DOCUMENTS No Limitations				
18. DEFTEST DESCRIPTORS Situation Awareness, Knowledge representation,Expert Systems,Java (Computer program language)				

19. ABSTRACT

SDL is a Situation Description Language intended for use in situation assessment problems. SDL provides knowledge modelling and inference facilities for reasoning with information.
This document describes a portable implementation of SDL in Java. It provides information required by a user of the system. Details include the operation of the compiler, the use of temporal knowledge and inference, and use of the visualisation system. This report also provides implementation details necessary for modifying or extending the system. A detailed example describes how the system was used for submarine situation assessment.